Halting problem undecidability and infinitely nested simulation

The halting theorem counter-examples present infinitely nested simulation (non-halting) behavior to every simulating halt decider. The pathological self-reference of the conventional halting problem proof counter-examples is overcome. The halt status of these examples is correctly determined. A simulating halt decider remains in pure simulation mode until after it determines that its input will never reach its final state. This eliminates the conventional feedback loop where the behavior of the halt decider effects the behavior of its input.

The x86utm operating system was created so that the halting problem could be examined concretely in the high level language of C. H is a function written in C that analyzes the x86 machine language execution trace of other functions written in C. H recognizes simple cases of infinite recursion and infinite loops. The conventional halting problem proof counter-example template is shown to simply be an input that does not halt.

H simulates its input with an x86 emulator until it determines that its input would never halt. As soon as H recognizes that its input would never halt it stops simulating this input and returns 0. For inputs that do halt H acts exactly as if it was an x86 emulator and simply runs its input to completion and then returns 1.

In theoretical computer science the random-access stored-program (RASP) machine model is an abstract machine used for the purposes of algorithm development and algorithm complexity theory. ...The RASP is closest of all the abstract models to the common notion of computer.

https://en.wikipedia.org/wiki/Random-access_stored-program_machine

The C/x86 model of computation is known to be Turing equivalent on the basis that it maps to the RASP model for all computations having all of the memory that they need. As long as an C/x86 function is a pure function of its inputs the C/x86 model of computation can be relied upon as a much higher level of abstraction of the behavior of actual Turing machines.

This criteria merely relies on the fact that the UTM simulation of a machine description of a machine is computationally equivalent to the direct execution of this same machine:

Simulating Halt Decider Theorem (Olcott 2020):
A simulating halt decider correctly decides that any input that never halts unless the simulating halt decider aborts its simulation of this input is an input that never halts.

the Turing machine halting problem. Simply stated, the problem is: given the description of a Turing machine M and an input w, does M, when started in the initial configuration q0w, perform a computation that eventually halts? (Linz:1990:317).

In computability theory, the halting problem is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running, or continue to run forever.

The halting problem is always about program descriptions not running programs. This means that it is always about the input to the halt decider not the direct execution of the program. If the input to the simulating halt decider never halts unless the halt decider aborts its simulation of this input then its input never halts.

Because H only acts as a pure simulator of its input until after its halt status decision has been made it has no behavior that can possibly effect the behavior of its input. Because of this H screens out its own address range in every execution trace that it examines. This is why we never see any instructions of H in any execution trace after an input calls H.

**Pathological Input** to a halt decider is stipulated to mean any input that was defined to do the opposite of whatever its corresponding halt decider decides as Sipser describes:

Now we construct a new Turing machine D with H as a subroutine. This new TM calls H to determine what M does when the input to M is its own description \( \langle M \rangle \). Once D has determined this information, it does the opposite. (Sipser:1997:165)

When D is invoked with input \( \langle D \rangle \) we have pathological self-reference when D calls H with \( \langle D \rangle \) and does the opposite of whatever H returns.

**Does D halt on its own machine description (D) ?**

This question can only be correctly answered after the pathology has been removed. When a halt decider only acts as a pure simulator of its input until after its halt status decision is made there is no feedback loop of back channel communication between the halt decider and its input that can prevent a correct halt status decision. In this case the halt decider is only examining the behavior of the input. It ignores it own behavior.

The standard pseudo-code halting problem template "proved" that the halting problem could never be solved on the basis that neither value of true (halting) nor false (not halting) could be correctly returned form the halt decider to the confounding input.

```plaintext
procedure compute_g(i): // (Wikipedia:Halting Problem)
    if f(i, i) == 0 then // adapted from (Strachey, C 1965)
        return 0 // originally written in CPL
    else // ancestor of the BCPL, B and C
        loop forever // programming languages
```

This problem is overcome on the basis that a simulating halt decider would abort the simulation of its input before ever returning any value to this input. It aborts the simulation of its input on the basis that its input specifies what is essentially infinite recursion (infinitely nested simulation) to any simulating halt decider.

Every input to a simulating halt decider that only stops running when its simulation is aborted unequivocally specifies a computation that never halts. When input to a simulating halt decider cannot possibly reach its final state then we know that this input never halts.
A simulating halt decider $H$ divides all of its input into:

(1) Those inputs that never halt unless $H$ aborts their simulation (never halting).
   $H$ aborts its simulation of these inputs an returns 0 for never halting.

(2) Those inputs that halt while $H$ remains a pure simulator (halting).
   $H$ waits for its simulation of this input to complete and then returns 1 halting.

$H$ derives the execution trace of its inputs as a pure function of these inputs.
$H$ derives its halt status decision as a pure function of this derived execution trace.
Therefore $H$ derives its halt status decision as a pure function of its inputs.
Simulating partial halt decider H correctly decides that P(P) never halts (V1)

H analyzes the (currently updated) stored execution trace of its x86 emulation of P(P) after it simulates each instruction of input (P, P). As soon as a non-halting behavior pattern is matched H aborts the simulation of its input and decides that its input never halts.

The execution trace of the x86 emulation of P(P) by simulating halt decider H conclusively proves that P never halts unless H aborts its simulation of P. This provides complete proof that the input to H never halts thus H(P,P)==0 is correct.

// Simplified Linz Ĥ (Linz:1990:319)
// Strachey(1965) CPL translated to C

void P(u32 x)
{
    if (H(x, x))
        HERE: goto HERE;
}

int main()
{
    Output("Input_Halts = ", H((u32)P, (u32)P));
}

// machine   stack     stack     machine    assembly
// address   address   data      code       language
//---------  ---------  ------  --------  =========
[00000c56]  [0010172a]  [00000000]  55          push ebp
[00000c57]  [0010172a]  [00000000]  8bec        mov ebp,esp
[00000c59]  [00101726]  [00000c36]  68360c0000  push 00000c36 // push P
[00000c5e]  [00101722]  [00000c36]  68360c0000  push 00000c36 // push P
[00000c63]  [0010171e]  [00000c68]  e8fecedfff  call 00000966 // call H(P,P)
[00000c68]  [0010171e]  [00000c6c]  83c408      add esp,+08
[00000c6b]  [0010171e]  [00000c6e]  50          push eax
[00000c6c]  [0010171e]  [00000357]  6857030000  push 00000357
[00000c71]  [0010171e]  [00000c76]  e8fecedfff  call 00000966 // call H(P,P)
[00000c76]  [0010171e]  [00000c7a]  83c408      add esp,+08
[00000c79]  [0010171e]  [00000c7b]  33c0        xor eax,eax
[00000c7b]  [0010171e]  [00000c7c]  c3          ret

Size in bytes:(0027) [00000c50]

// main
[00000c56]  [0010172a]  [00000000]  55          push ebp
[00000c57]  [0010172a]  [00000000]  8bec        mov ebp,esp
[00000c59]  [00101726]  [00000c36]  68360c0000  push 00000c36 // push P
[00000c5e]  [00101722]  [00000c36]  68360c0000  push 00000c36 // push P
[00000c63]  [0010171e]  [00000c68]  e8fecedfff  call 00000966 // call H(P,P)
[00000c68]  [0010171e]  [00000c6c]  83c408      add esp,+08
[00000c6b]  [0010171e]  [00000c6e]  50          push eax
[00000c6c]  [0010171e]  [00000357]  6857030000  push 00000357
[00000c71]  [0010171e]  [00000c76]  e8fecedfff  call 00000966 // call H(P,P)
[00000c76]  [0010171e]  [00000c7a]  83c408      add esp,+08
[00000c79]  [0010171e]  [00000c7b]  33c0        xor eax,eax
[00000c7b]  [0010171e]  [00000c7c]  c3          ret

Size in bytes:(0039) [00000c7c]
Begin Local Halt Decider Simulation at Machine Address:c36

We do not see any of the x86 instructions of H in the above execution trace because we know that H is only acting as a pure simulator of its inputs until after it has made its halt status decision. This means that H cannot possibly have any effect on the behavior of its input during the above (execution trace / halt status analysis), thus H can safely ignore its own instructions in this halt status analysis.

It required that we make sure to ignore the behavior of H in the execution trace so that we eliminate the pathological self-reference error from the halt status analysis.

We can verify that the above execution trace of the simulation of P(P) is accurate on the basis that it precisely corresponds to the x86 assembly language source-code of P.

This infinite recursion detection criteria are met by the above execution trace:
(a) P calls H twice in sequence from the same machine address.
(b) With the same parameters: (P,P) to H.
(c) With no conditional branch or indexed jump instructions in the execution trace of P.
(d) We know that there are no return instructions in H because we know that H is in pure simulation mode.

This conclusively proves that P never halts unless H aborts its simulation of P which proves that the behavior of the simulation of P on input P by H meets the following criteria:

Simulating Halt Decider Theorem (Olcott 2020):
A simulating halt decider correctly decides that any input that never halts unless the simulating halt decider aborts its simulation of this input is an input that never halts.

This criteria merely relies on the fact that the UTM simulation of a machine description of a machine is computationally equivalent to the direct execution of this same machine:

---5---
2021-09-26 09:39 AM
The direct execution of P(P) halts (V2)

The execution trace of the x86 emulation of P(P) by simulating halt decider H conclusively proves that P cannot possibly ever reach its final state of 0xc3f. This provides complete proof that that the input to H never halts thus \( H(P,P) = 0 \) is correct.

// Simplified Linz Ĥ (Linz:1990:319)  
// Strachey(1965) CPL translated to C  
void P(u32 x)  
{  
  if (H(x, x))  
      HERE: goto HERE;  
}  

int main()  
{  
  P((u32)P);  
}  

---  
6  
---

2021-09-26 09:39 AM
Begin Local Halt Decider Simulation at Machine Address: c25

Number of User Instructions (34)
Number of Instructions Executed (23729)

// Simplified Linz Ĥ (Linz:1990:319)
// Strachey(1965) CPL translated to C
void P(u32 x)
{
    if (H(x, x))
        HERE: goto HERE;
}

int main()
{
    H((u32)P, (u32)P);
    P((u32)P);
}

Because the first line of main() executes H(P,P) first and then P(P) and the second line of main() executes P(P) first and then H(P,P) the difference in the relative execution order makes this pair of computations distinctly different computations. Distinctly different computations can have different behavior without contradiction.
Simulating partial halt decider H1 correctly decides that P(P) halts (V3)

When we create an exact copy H1 of H and invoke H1(P,P) in main() it can see that H aborts its simulation of its input thus H1 returns 1 indicating that its input halts.

When H1(P,P) is invoked in main() its invocation order is P(P) then H(P,P).
When P(P) is invoked in main() its invocation order is P(P) then H(P,P).
Because the same invocation order is preserved H1 provides a halt status decision that is consistent with the behavior of P(P) invoked in main().

H1(P,P) is a distinctly different computation than H(P,P) because H1 can see what H(P,P) does yet H(P,P) cannot see anything that H1(P,P) does.

The master UTM / halt decider H1 can see everything that it simulates:
(a) P begins
(b) P calls H(P,P)
(c) H returns to P
(d) P returns to main() where it was simulated by H1.

The master UTM / halt decider H can see everything that it simulates:
(a) P begins
(b) P calls H(P,P)
(c) P begins
(e) P calls H(P,P)

H(P,P) sees that it must abort its simulation of its input or its input never halts.

```c
void P(u32 x)
{
    if (H(x, x))
        HERE: goto HERE;
}

int main()
{
    Output("Input_Halts = ", H1((u32)P, (u32)P));
}
```

x86 assembly language source-code for the above C functions.

```
P()
[00000e52](01)  55          push ebp
[00000e53](02)  8bec        mov ebp,esp
[00000e55](03)  8b4508      mov eax,[ebp+08]
[00000e58](01)  50          push eax
[00000e59](03)  8b4d08      mov ecx,[ebp+08]
[00000e5c](01)  51          push ecx
[00000e5d](05)  e870feffff  call 00000cd2 // call H
[00000e62](03)  83c408      add esp,+08
[00000e65](02)  85c0        test eax,eax
[00000e67](02)  7402        jz 00000e6b // jmp if eax == 0
[00000e69](02)  ebfe        jmp 00000e69 // eax != 0
[00000e6b](01)  5d          pop ebp
[00000e6c](01)  c3          ret
Size in bytes:(0027) [00000e6c]
```
Even though H1 has identical code to H it has different behavior than H because it is executed first. Each master UTM / halt decider can only see the user-code instructions that it simulates. This includes the user-code that is simulated by recursive simulations of itself. H can see lines 13-26. H1 can see lines 6-12 and lines 27-31. H1 returns to main() at line 31.

All halt deciders ignore analyzing the execution trace of any operating system source-code because all operating system code is known to halt. The halt deciders themselves are operating system functions.

---9---

2021-09-26 09:39 AM
Simulating partial halt decider $H$ correctly decides that $\text{Infinite\_Loop()}$ never halts

```c
void Infinite\_Loop()
{
    \textbf{HERE: goto HERE;}
}

int main()
{
    u32 Input\_Would\_Halt2 = H((u32)\text{Infinite\_Loop}, (u32)\text{Infinite\_Loop});
    Output("Input\_Would\_Halt2 = ", Input\_Would\_Halt2);
}
```

Number of User Instructions (39)
Execution Trace of H(Infinite_Loop, Infinite_Loop)

<table>
<thead>
<tr>
<th>machine address</th>
<th>stack address</th>
<th>stack data</th>
<th>machine code</th>
<th>assembly language</th>
</tr>
</thead>
<tbody>
<tr>
<td>[00000c00]</td>
<td>[00101693]</td>
<td>0000000000</td>
<td>55</td>
<td>push ebp</td>
</tr>
<tr>
<td>[00000c01]</td>
<td>[00101693]</td>
<td>0000000000</td>
<td>8bec</td>
<td>mov ebp,esp</td>
</tr>
<tr>
<td>[00000c03]</td>
<td>[0010168f]</td>
<td>0000000000</td>
<td>51</td>
<td>push ecx</td>
</tr>
<tr>
<td>[00000c04]</td>
<td>[0010168b]</td>
<td>000000ab0</td>
<td>68b00a0000</td>
<td>push 00000ab0</td>
</tr>
<tr>
<td>[00000c09]</td>
<td>[00101687]</td>
<td>000000ab0</td>
<td>68b00a0000</td>
<td>push 00000ab0</td>
</tr>
<tr>
<td>[00000c0e]</td>
<td>[00101683]</td>
<td>000000c13</td>
<td>e84dfdffff</td>
<td>call 00000960</td>
</tr>
</tbody>
</table>

Begin Local Halt Decider Simulation at Machine Address: ab0

<table>
<thead>
<tr>
<th>machine address</th>
<th>stack address</th>
<th>stack data</th>
<th>machine code</th>
<th>assembly language</th>
</tr>
</thead>
<tbody>
<tr>
<td>[00000ab0]</td>
<td>[00211733]</td>
<td>00211737</td>
<td>55</td>
<td>push ebp</td>
</tr>
<tr>
<td>[00000ab1]</td>
<td>[00211733]</td>
<td>00211737</td>
<td>8bec</td>
<td>mov ebp,esp</td>
</tr>
<tr>
<td>[00000ab3]</td>
<td>[00211733]</td>
<td>00211737</td>
<td>ebfe</td>
<td>jmp 00000ab3</td>
</tr>
<tr>
<td>[00000ab3]</td>
<td>[00211733]</td>
<td>00211737</td>
<td>ebfe</td>
<td>jmp 00000ab3</td>
</tr>
</tbody>
</table>

Local Halt Decider: Infinite Loop Detected Simulation Stopped

<table>
<thead>
<tr>
<th>machine address</th>
<th>stack address</th>
<th>stack data</th>
<th>machine code</th>
<th>assembly language</th>
</tr>
</thead>
<tbody>
<tr>
<td>[00000c13]</td>
<td>[0010168f]</td>
<td>0000000000</td>
<td>83c408</td>
<td>add esp,+08</td>
</tr>
<tr>
<td>[00000c16]</td>
<td>[0010168f]</td>
<td>0000000000</td>
<td>8945fc</td>
<td>mov [ebp-04],eax</td>
</tr>
<tr>
<td>[00000c19]</td>
<td>[0010168f]</td>
<td>0000000000</td>
<td>8b45fc</td>
<td>mov eax,[ebp-04]</td>
</tr>
<tr>
<td>[00000c1c]</td>
<td>[0010168b]</td>
<td>0000000000</td>
<td>50</td>
<td>push eax</td>
</tr>
<tr>
<td>[00000c1d]</td>
<td>[00101687]</td>
<td>0000034b</td>
<td>684b030000</td>
<td>push 0000034b</td>
</tr>
<tr>
<td>[00000c22]</td>
<td>[00101687]</td>
<td>0000034b</td>
<td>e859f7ffff</td>
<td>call 00000380</td>
</tr>
</tbody>
</table>

Input_Would_Halt2 = 0

<table>
<thead>
<tr>
<th>machine address</th>
<th>stack address</th>
<th>stack data</th>
<th>machine code</th>
<th>assembly language</th>
</tr>
</thead>
<tbody>
<tr>
<td>[00000c27]</td>
<td>[0010168f]</td>
<td>0000000000</td>
<td>83c408</td>
<td>add esp,+08</td>
</tr>
<tr>
<td>[00000c2a]</td>
<td>[0010168f]</td>
<td>0000000000</td>
<td>33c0</td>
<td>xor eax,eax</td>
</tr>
<tr>
<td>[00000c2c]</td>
<td>[00101693]</td>
<td>0000000000</td>
<td>8be5</td>
<td>mov esp,ebp</td>
</tr>
<tr>
<td>[00000c2e]</td>
<td>[00101697]</td>
<td>0010000000</td>
<td>5d</td>
<td>pop ebp</td>
</tr>
<tr>
<td>[00000c2f]</td>
<td>[0010169b]</td>
<td>000000050</td>
<td>c3</td>
<td>ret</td>
</tr>
</tbody>
</table>

Number of User Instructions(21)
Number of Instructions Executed(640)
Simulating partial halt decider H decides that Infinite_Recursion() never halts

```c
void Infinite_Recursion(u32 N)
{
    Infinite_Recursion(N);
}

int main()
{
    u32 Input_Halts = H((u32)Infinite_Recursion, 3);
    Output("Input_Halts = ", Input_Halts);
}
```

---

2021-09-26 09:39 AM
Begin Local Halt Decider Simulation at Machine Address:ac6

Infinite Recursion Detection Criteria:

- Infinite_Recursion() calls itself recursively with the same input. It has no escape from this infinite recursion. H recognizes this infinite behavior pattern, aborts its simulation of Infinite_Recursion() and reports that this input never halts.

Infinite recursion detection criteria:

- If the execution trace of function X() called by function Y() shows:
  1. Function X() is called twice in sequence from the same machine address of Y().
  2. With the same parameters to X().
  3. With no conditional branch or indexed jump instructions in Y().
  4. With no function call returns from X().

then the function call from Y() to X() is infinitely recursive.
Simulating partial halt decider $H$ decides that $\text{Factorial}(3)$ halts

```c
int Factorial(int n)
{
    Output("Factorial(n)",n);
    if (n > 1)
        return n * Factorial(n - 1);
    else
        return 1;
}

int main()
{
    Output("Input_Halts = ", H(Factorial, 3));
}
```

---

2021-09-26  09:39 AM

---14---
<table>
<thead>
<tr>
<th>machine address</th>
<th>stack address</th>
<th>stack data</th>
<th>machine code</th>
<th>assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>[000000ea2]</td>
<td>[00101ae7]</td>
<td>[00000000]</td>
<td>55</td>
<td>push ebp</td>
</tr>
<tr>
<td>[000000ea3]</td>
<td>[00101ae7]</td>
<td>[00000000]</td>
<td>8bec</td>
<td>mov ebp, esp</td>
</tr>
<tr>
<td>[000000ea5]</td>
<td>[00101ae3]</td>
<td>[00000000]</td>
<td>6a03</td>
<td>push +03</td>
</tr>
<tr>
<td>[000000ea7]</td>
<td>[00101ad3]</td>
<td>[000000de2]</td>
<td>68e20d0000</td>
<td>push 00000de2</td>
</tr>
<tr>
<td>[000000ea9]</td>
<td>[00101adb]</td>
<td>[00000e0b1]</td>
<td>e821feffff</td>
<td>call 00000cd2</td>
</tr>
</tbody>
</table>

Begin Local Halt Decider Simulation at Machine Address:de2

<table>
<thead>
<tr>
<th>machine address</th>
<th>stack address</th>
<th>stack data</th>
<th>machine code</th>
<th>assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>[000000de2]</td>
<td>[00211b8d]</td>
<td>[00211b8b]</td>
<td>55</td>
<td>push ebp</td>
</tr>
<tr>
<td>[000000de3]</td>
<td>[00211b8d]</td>
<td>[00211b8b]</td>
<td>8bec</td>
<td>mov ebp, esp</td>
</tr>
<tr>
<td>[000000de5]</td>
<td>[00211b8d]</td>
<td>[00211b8b]</td>
<td>8b4508</td>
<td>mov eax,[ebp+08]</td>
</tr>
<tr>
<td>[000000de8]</td>
<td>[00211b8e3]</td>
<td>[00000003]</td>
<td>50</td>
<td>push eax</td>
</tr>
<tr>
<td>[000000de9]</td>
<td>[00211b7f]</td>
<td>[00000313]</td>
<td>6813030000</td>
<td>push 00000313</td>
</tr>
<tr>
<td>[000000dee]</td>
<td>[00211b7f]</td>
<td>[00000313]</td>
<td>e85ff5ffff</td>
<td>call 00000352</td>
</tr>
</tbody>
</table>

Factorial(n)3

<table>
<thead>
<tr>
<th>machine address</th>
<th>stack address</th>
<th>stack data</th>
<th>machine code</th>
<th>assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>[000000df3]</td>
<td>[00211b87]</td>
<td>[00211b8b]</td>
<td>83c408</td>
<td>add esp,+08</td>
</tr>
<tr>
<td>[000000df6]</td>
<td>[00211b87]</td>
<td>[00211b8b]</td>
<td>837d0801</td>
<td>cmp dword [ebp+08],+01</td>
</tr>
<tr>
<td>[000000df9]</td>
<td>[00211b87]</td>
<td>[00211b8b]</td>
<td>8b4d08</td>
<td>mov ecx,[ebp+08]</td>
</tr>
<tr>
<td>[000000dfc]</td>
<td>[00211b87]</td>
<td>[00211b8b]</td>
<td>83e901</td>
<td>sub ecx,+01</td>
</tr>
<tr>
<td>[000000deo]</td>
<td>[00211b88]</td>
<td>[00000002]</td>
<td>51</td>
<td>push ecx</td>
</tr>
<tr>
<td>[000000def]</td>
<td>[00211b7f]</td>
<td>[00000e08]</td>
<td>e8dafffffff</td>
<td>call 00000de2</td>
</tr>
<tr>
<td>[000000deg]</td>
<td>[00211b7b]</td>
<td>[00211b87]</td>
<td>55</td>
<td>push ebp</td>
</tr>
<tr>
<td>[000000deh]</td>
<td>[00211b7b]</td>
<td>[00211b87]</td>
<td>8bec</td>
<td>mov ebp, esp</td>
</tr>
<tr>
<td>[000000dei]</td>
<td>[00211b7b]</td>
<td>[00211b87]</td>
<td>8b4508</td>
<td>mov eax,[ebp+08]</td>
</tr>
<tr>
<td>[000000dje]</td>
<td>[00211b77]</td>
<td>[00000002]</td>
<td>50</td>
<td>push eax</td>
</tr>
<tr>
<td>[000000dof]</td>
<td>[00211b73]</td>
<td>[00000313]</td>
<td>6813030000</td>
<td>push 00000313</td>
</tr>
<tr>
<td>[000000deg]</td>
<td>[00211b7b]</td>
<td>[00000313]</td>
<td>e85ff5ffff</td>
<td>call 00000352</td>
</tr>
</tbody>
</table>

Factorial(n)2

<table>
<thead>
<tr>
<th>machine address</th>
<th>stack address</th>
<th>stack data</th>
<th>machine code</th>
<th>assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>[000000df3]</td>
<td>[00211b7b]</td>
<td>[00211b87]</td>
<td>83c408</td>
<td>add esp,+08</td>
</tr>
<tr>
<td>[000000df6]</td>
<td>[00211b7b]</td>
<td>[00211b87]</td>
<td>837d0801</td>
<td>cmp dword [ebp+08],+01</td>
</tr>
<tr>
<td>[000000df9]</td>
<td>[00211b7b]</td>
<td>[00211b87]</td>
<td>8b4d08</td>
<td>mov ecx,[ebp+08]</td>
</tr>
<tr>
<td>[000000dfc]</td>
<td>[00211b7b]</td>
<td>[00211b87]</td>
<td>83e901</td>
<td>sub ecx,+01</td>
</tr>
<tr>
<td>[000000deo]</td>
<td>[00211b77]</td>
<td>[00000001]</td>
<td>51</td>
<td>push ecx</td>
</tr>
<tr>
<td>[000000def]</td>
<td>[00211b73]</td>
<td>[00000e08]</td>
<td>e8dafffffff</td>
<td>call 00000de2</td>
</tr>
<tr>
<td>[000000deg]</td>
<td>[00211b6f]</td>
<td>[00211b7b]</td>
<td>55</td>
<td>push ebp</td>
</tr>
<tr>
<td>[000000deh]</td>
<td>[00211b7b]</td>
<td>[00211b87]</td>
<td>8bec</td>
<td>mov ebp, esp</td>
</tr>
<tr>
<td>[000000dei]</td>
<td>[00211b7b]</td>
<td>[00211b87]</td>
<td>8b4508</td>
<td>mov eax,[ebp+08]</td>
</tr>
<tr>
<td>[000000dje]</td>
<td>[00211b6b]</td>
<td>[00000001]</td>
<td>50</td>
<td>push eax</td>
</tr>
<tr>
<td>[000000dof]</td>
<td>[00211b67]</td>
<td>[00000313]</td>
<td>6813030000</td>
<td>push 00000313</td>
</tr>
<tr>
<td>[000000deg]</td>
<td>[00211b67]</td>
<td>[00000313]</td>
<td>e85ff5ffff</td>
<td>call 00000352</td>
</tr>
</tbody>
</table>

Factorial(n)1

<table>
<thead>
<tr>
<th>machine address</th>
<th>stack address</th>
<th>stack data</th>
<th>machine code</th>
<th>assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>[000000df3]</td>
<td>[00211b7b]</td>
<td>[00211b87]</td>
<td>83c408</td>
<td>add esp,+08</td>
</tr>
<tr>
<td>[000000df6]</td>
<td>[00211b7b]</td>
<td>[00211b87]</td>
<td>837d0801</td>
<td>cmp dword [ebp+08],+01</td>
</tr>
<tr>
<td>[000000df9]</td>
<td>[00211b7b]</td>
<td>[00211b87]</td>
<td>8b4d08</td>
<td>mov ecx,[ebp+08]</td>
</tr>
<tr>
<td>[000000dfc]</td>
<td>[00211b7b]</td>
<td>[00211b87]</td>
<td>83e901</td>
<td>sub ecx,+01</td>
</tr>
<tr>
<td>[000000deo]</td>
<td>[00211b77]</td>
<td>[00000001]</td>
<td>51</td>
<td>push ecx</td>
</tr>
<tr>
<td>[000000def]</td>
<td>[00211b73]</td>
<td>[00000e08]</td>
<td>e8dafffffff</td>
<td>call 00000de2</td>
</tr>
<tr>
<td>[000000deg]</td>
<td>[00211b6f]</td>
<td>[00211b7b]</td>
<td>55</td>
<td>push ebp</td>
</tr>
<tr>
<td>[000000deh]</td>
<td>[00211b7b]</td>
<td>[00211b87]</td>
<td>8bec</td>
<td>mov ebp, esp</td>
</tr>
<tr>
<td>[000000dei]</td>
<td>[00211b7b]</td>
<td>[00211b87]</td>
<td>8b4508</td>
<td>mov eax,[ebp+08]</td>
</tr>
<tr>
<td>[000000dje]</td>
<td>[00211b6b]</td>
<td>[00000001]</td>
<td>50</td>
<td>push eax</td>
</tr>
<tr>
<td>[000000dof]</td>
<td>[00211b67]</td>
<td>[00000313]</td>
<td>6813030000</td>
<td>push 00000313</td>
</tr>
<tr>
<td>[000000deg]</td>
<td>[00211b67]</td>
<td>[00000313]</td>
<td>e85ff5ffff</td>
<td>call 00000352</td>
</tr>
</tbody>
</table>

Input_Halts = 1

<table>
<thead>
<tr>
<th>machine address</th>
<th>stack address</th>
<th>stack data</th>
<th>machine code</th>
<th>assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>[000000eaf]</td>
<td>[00101aef]</td>
<td>[00000000]</td>
<td>83c408</td>
<td>add esp,+08</td>
</tr>
<tr>
<td>[000000eac]</td>
<td>[00101aeb]</td>
<td>[00000000]</td>
<td>33c0</td>
<td>xor eax,eax</td>
</tr>
<tr>
<td>[000000ecc]</td>
<td>[00101aeh]</td>
<td>[00000000]</td>
<td>5d</td>
<td>pop ebp</td>
</tr>
<tr>
<td>[000000ec5]</td>
<td>[00101aef]</td>
<td>[000000c8]</td>
<td>c3</td>
<td>ret</td>
</tr>
</tbody>
</table>

Number of User Instructions(51)
Number of Instructions Executed(3714)
Strachey’s Impossible Program

To the Editor,
The Computer Journal.

An impossible program

Sir,
A well-known piece of folk-lore among programmers holds that it is impossible to write a program which can examine any other program and tell, in every case, if it will terminate or get into a closed loop when it is run.
I have never actually seen a proof of this in print, and though Alan Turing once gave me a verbal proof (in a railway carriage on the way to a Conference at the NPL in 1953), I unfortunately and promptly forgot the details. This left me with an uneasy feeling that the proof must be long or complicated, but in fact it is so short and simple that it may be of interest to casual readers. The version below uses CPL, but not in any essential way.

Suppose T[R] is a Boolean function taking a routine (or program) R with no formal or free variables as its argument and that for all R, T[R] — True if R terminates if run and that T[R] = False if R does not terminate.
Consider the routine P defined as follows

```
rec routine P
  §L: if T[P] go to L
  Return §
```

If T[P] = True the routine P will loop, and it will only terminate if T[P] = False. In each case T[P] has exactly the wrong value, and this contradiction shows that the function T cannot exist.

Yours faithfully,
C. STRACHEY.

Churchill College,
Cambridge.

Peter Linz Ĥ applied to the Turing machine description of itself: \(〈\hat{H}〉\)

The following simplifies the syntax for the definition of the Linz Turing machine Ĥ, it is now a single machine with a single start state. A simulating halt decider is embedded at Ĥ.qx. It has been annotated so that it only shows Ĥ applied to \(〈\hat{H}〉\), converting the variables to constants.

\[
\hat{H}.q_0 (〈\hat{H}〉) \rightarrow^* \hat{H}.q_x (〈\hat{H}〉) (q_0) \rightarrow^* \hat{H}.q_y \infty
\]

if the simulated \(〈\hat{H}〉\) applied to \(〈\hat{H}〉\) halts, and

\[
\hat{H}.q_0 (〈\hat{H}〉) \rightarrow^* \hat{H}.q_x (〈\hat{H}〉) (q_0) \rightarrow^* \hat{H}.q_n
\]

if the simulated \(〈\hat{H}〉\) applied to \(〈\hat{H}〉\) does not halt

![Figure 12.3 Turing Machine Ĥ applied to \(〈\hat{H}〉\)](image)

\(\hat{H}\) copies its input \(〈\hat{H}_1〉\) to \(〈\hat{H}_2〉\) then simulates this input \(\hat{H}_1\) with its input \(〈\hat{H}_2〉\) which copies its input \(〈\hat{H}_3〉\) to \(〈\hat{H}_4〉\) then simulates this input \(\hat{H}_2\) with its input \(〈\hat{H}_3〉\) which copies its input \(〈\hat{H}_5〉\) to \(〈\hat{H}_6〉\) then simulates this input \(\hat{H}_3\) with its input \(〈\hat{H}_4〉\)...

Until the simulating halt decider at \(\hat{H}.q_x\) aborts the simulation of its input this input never halts. Even though this repeating pattern is more complex (because it copies its input) than the above x86 example of `int main() { H((u32)P, (u32)P); }` it is still a repeating pattern that can be recognized by a simulating halt decider.

The transition from \(\hat{H}.q_x\) to \(\hat{H}.q_0\) expresses the gist of the idea of infinitely nested simulation. It is not the conventional notion of a state transition within the same machine instance.

![Figure 12.4 Turing Machine Ĥ applied to \(〈\hat{H}〉\) input](image)
When the original Linz H is applied to \( \langle \hat{H} \rangle \langle \hat{H} \rangle \) it sees that its input transitions to \( \hat{H}.qn \). This provides the basis for H to transition to its final state of H.qy.

When \( \hat{H}.qx \) is applied to \( \langle \hat{H} \rangle \langle \hat{H} \rangle \) it sees that none of the recursive simulations of its input ever halt so it transitions to its final state of \( \hat{H}.qn \).

The master slave relationship from H to \( \hat{H}.qx \) makes them distinctly different computations even though they are otherwise identical and have the same input.

When a separate halt decider H is applied to its input it correctly decides that this input halts. 

```c
int main() { H1(P,P); }
```

correctly decides that its input halts.

When an input is defined using the same halt decider that is applied to this input this pathological self-reference error can be detected and rejected on the basis that

```c
int main()
{
    if (H1((u32)P, (u32)P) != H((u32)P, (u32)P))
        OutputString("Pathological self-reference error!");
}
```

Copyright 2016-2021 PL Olcott

**Strachey, C 1965.** An impossible program The Computer Journal, Volume 7, Issue 4, January 1965, Page 313, [https://doi.org/10.1093/comjnl/7.4.313](https://doi.org/10.1093/comjnl/7.4.313)

**Linz, Peter 1990.** An Introduction to Formal Languages and Automata. Lexington/Toronto: D. C. Heath and Company. (318-320)

There does not exist any Turing machine $H$ that behaves as required by Definition 12.1. The halting problem is therefore undecidable.

**Proof:** We assume the contrary, namely that there exists an algorithm, and consequently some Turing machine $H$, that solves the halting problem. The input to $H$ will be the description (encoded in some form) of $M$, say $w_M$, as well as the input $w$. The requirement is then that, given any $(w_M, w)$, the Turing machine $H$ will halt with either a yes or no answer. We achieve this by asking that $H$ halt in one of two corresponding final states, say, $q_y$ or $q_n$. The situation can be visualized by a block diagram like Figure 12.1. The intent of this diagram is to indicate that, if $M$ is started in state $q_0$ with input $(w_M, w)$, it will eventually halt in state $q_y$ or $q_n$. As required by Definition 12.1, we want $H$ to operate according to the following rules:

$$q_0 w_M w \xrightarrow{H} q_y x_1 x_2,$$

if $M$ applied to $w$ halts, and

$$q_0 w_M w \xrightarrow{H} q_n y_1 y_2,$$

if $M$ applied to $w$ does not halt.

**Figure 12.1**
Next, we modify $H$ to produce a Turing machine $H'$ with the structure shown in Figure 12.2. With the added states in Figure 12.2 we want to convey that the transitions between state $q_y$ and the new states $q_a$ and $q_b$ are to be made, regardless of the tape symbol, in such a way that the tape remains unchanged. The way this is done is straightforward. Comparing $H$ and $H'$ we see that, in situations where $H$ reaches $q_y$ and halts, the modified machine $H'$ will enter an infinite loop. Formally, the action of $H'$ is described by

$$q_0 w_M w \xrightarrow{*} H' \infty,$$

if $M$ applied to $w$ halts, and

$$q_0 w_M w \xrightarrow{*} H' y_1 q_n y_2,$$

if $M$ applied to $w$ does not halt.

From $H'$ we construct another Turing machine $\hat{H}$. This new machine takes as input $w_M$, copies it, and then behaves exactly like $H'$. Then the action of $\hat{H}$ is such that

$$q_0 w_M \xrightarrow{*} \hat{H} q_0 w_M w_M \xrightarrow{*} \hat{H} \infty,$$

if $M$ applied to $w_M$ halts, and

$$q_0 w_M \xrightarrow{*} \hat{H} q_0 w_M w_M \xrightarrow{*} \hat{H} y_1 q_n y_2,$$

if $M$ applied to $w_M$ does not halt.
Now $\hat{H}$ is a Turing machine, so that it will have some description in $\Sigma^*$, say $\hat{w}$. This string, in addition to being the description of $\hat{H}$ can also be used as input string. We can therefore legitimately ask what would happen if $\hat{H}$ is applied to $\hat{w}$. From the above, identifying $M$ with $\hat{H}$, we get

$$q_0\hat{w} \vdash^{*} \hat{H}\infty,$$

if $\hat{H}$ applied to $\hat{w}$ halts, and

$$q_0\hat{w} \vdash^{*} \hat{H} y_1 q_n y_2,$$

if $\hat{H}$ applied to $\hat{w}$ does not halt. This is clearly nonsense. The contradiction tells us that our assumption of the existence of $H$, and hence the assumption of the decidability of the halting problem, must be false. ■